The design of a LISP system on 8 bit micro-computers

Jérôme CHAILLOUX

ABSTRACT :

We describe in this paper, the design of a LISP system on 8 bit micro-computers and its incarnation : the VLISP 8 interpreter, implemented on INTEL 8080 based systems.
According to the space and time limitations on such computers, several representations and manipulations of (atypical) LISP objects are discussed.
VLISP 8, which is compatible with other VLISP systems, has a new style of function invocation (in using of F-TYP and F-VAL attributes), a management of small integer numbers which don't need any extra storage and an evaluator which does not create any internal Cons-Cells.

1.0 Introduction

Since 1971 numerous LISP interpreters [McCarthy 1962, Allen 1978] have been implemented at the Université de Paris VIII - Vincennes on various machines, CAE510 [Greussay 1972], CAB500 [Wertz 1974], T1600 and SOLAR 16 [Greussay 1978], PDP10 [Chailloux 1978b].

Since then a new dialect of LISP, called VLISP, has been developped and formalized [Greussay 1977]. Some other new implementations are planed in particular on the PDP11 and on the T.I. 9900.

VLISP interpreters are designed to be implemented on small machines and are extremely fast. VLISP 8, its last incarnation, has been especially built for 8 bit micro-computers.

The VLISP 8 interpreter is presently available on the following systems :
- ISIS I and II (system of the INTEL MDS80, a 8080 based system)
- DDT 80 (minimum system of the MOSTEK SDB 80E, a Zilog 80 based system)

Our Zilog 80 system has been built by L. Audoire at the Université de Vincennes. He has added some new devices such as :
- a fast arithmetic processing unit
- an address bus demon connected onto the interrupt system
- an interface with COLORIX, a color TV screen device [Audoire 1976].

The interpreter occupies 8k bytes (they may be in ROM) and is easily transported on all other 8080 based systems; a minimum of 16k RAM is needed to store user created LISP objects.
The following paragraphes will discuss different questions related to the internal representations of VLISP objects and to the implementation of VLISP interpreter on micro-computer.

2.0 VLISP objects

All the LISP interpreters can handle objects as literal atoms, numbers and lists. Some implementations on large computers (PDP10, UNIVAC, IRIS80 ...) manipulate also objects such as character strings, arrays (of any kind), buffers, queues ...
VLISP 8 has retained only the three basic types : literal atoms, numbers and lists.
Therefore, some functions using the P-name of atoms allow the use of characters, and direct access to memory permits the manipulation of binary arrays.

2.1 Memory organisation

To distinguish very easily objects' LISP type and to facilitate the work of the Garbage-collection, memory has been sliced into fixed areas :

```
address                     pointers to memory

          ----------------
/0000    |                |
         |  interpreter   |
         |      area      |
         |................|                    Integer
         |                |
  16k    |     stack      |                    numbers
 bytes   |................|
                                                 area
```

```
        |...............|
        |               | <- HCODE
        |   code area   |
        | (used by the  |
        |   compiler)   |
        |_______________|

/4000    _______________
        |               | <- HNUMB
        | floating point|
        |  numbers area |
        |...............|
        |               | <- HNIL            Other
        | the NIL atom  |
        |...............|                    atoms
        |               |
 16k    |     other     |                    area
 bytes  |   constants   |
        | T EXPR etc .. |
        |...............|
        |               | <- HVAR
        | literal atoms |
        |     area      |
        |_______________|
/8000    _______________
        |               | <- HLIST           List
 32k    |  Cons-cells   |                    area
 bytes  |     area      |
        |_______________|
```

2.2 Type tests

It is crucial to test very quickly the type of a pointer. Indeed statistics about the working of a VLISP interpreter, written for the VCMC1 machine [Chailloux 1978a] (the ancestor of the VCMC2 machine used by the compiler), have shown us that nearly 20% of intructions executed in an interpreter were type test instructions.

The fixed areas organisation of the memory enable to carry out type tests very quickly since a type becomes an address function.

Because of the absence of a 16 bit comparator, the memory slices are on H borders (i.e. aligned on 256 byte pages). This slicing is easy to implement for the large areas but forces to use one entire page to represent often referenced objects (such as NIL or other constants).

ex: test of the pointer contained in HL

```
        MOV     A,H
        ORA     A
        JM      if HL is a list

        MOV     A,H
        CPI     HNIL
        JZ      if HL is NIL

        MOV     A,H
        CPI     HNUMB
        JC      if HL is an integer number
```

The main lack of this organisation is that all the available memory is not shared between the different areas, and the system is completely stopped as soon as one area is full.

To remedy this, a BIBOP-like organisation (for Big Bag of Pages) "à la MACLISP" [Steele 1974, Baker 1977b] has been considered; it consists of slicing the memory into several pages, each page occuping 256 bytes and having a type coded in a special pages' type table.
To test such a pointer we can use :

```
MOV     C,H     ; move the page number.
MVI     B,HATTP ; load the address of the pages' type table.
LDAX    BC      ; load A with the type of the page
                ; NOW the type can be tested.
```

This solution is not presently implemented in any available implementation for reasons of rapidity and large complications for the Gabage-collection.

2.3 The stack

VLISP is designed to use only one stack, combining data-stack and control-stack.
The 8080, with a stack in memory and a powerful instruction set using it, allows very powerful and pleasant stack manipulations. Particularily we can note the exchange of the top of stack with the register pair HL. This instruction (XTHL) is coded on 8 bits!
Unfortunately, the use of the stack pointer itself (store, comparison ...) is not easy. The stack overflow test, necessary in the recursive functions of the interpreter (i.e. READ, PRINT, EVAL, EQUAL ...) and in all compiled functions, is coded in the following way :

```
LXI     HL,-<top of stack>
DAD     SP
JC      <stack overflow>
```

which requires 30 clock cycles and destroys the main register pair HL. To prevent this, a special hardware device, has been added on our Zilog 80 system. A "demon" (which is just a hardware comparator) is placed on the address bus. This demon enables an interrupt if the address on the address bus is the same as the one which has been programmed at the beginning of the interpreter execution. It is a very convenient means to trap the stack overflows.

2.4 Literal atoms

These are the identifiers of the language and are used to name variables, functions, labels ... These atoms are represented by a sequence of any sort of characters. Only, the 127 first characters are significant. To include some special characters (like separators) we have to suround the whole P-name by " or to prefixe each litigious character by /. Atoms enclosed by " are considered like constants (i.e. they have themself for value). This feature allows the compatibility with others interpreters manipulating character strings.

A literal atom is represented by a pointer to a block of attributes stored in the special area of literal atoms.

Attributes are :

C-VAL (abbreviation of Cell-Value) contains, at any time, the VLISP value bound to the atom considered as a variable. This value (which is stored in the CAR of the atom) is accessible very quickly (cf: variable binding). At its creation time, the value

of an atom is the special indicator UNDEFINED, which, when evaluated, activates the UNDEFINED VARIABLE ERROR.

P-LIST (abbreviation of Property List) contains the property list of the atom. These properties are controlled by the user with the special functions (PUTPROP, GETPROP ...) except the property FUNCTION which is controlled by the interpreter (cf: function definition). The P-list of an atom is stored in the CDR of the atom.

F-VAL (abbreviation of Function Value) contains, at any time, the value associated with the atom considered as a function. This value can be :
- an address in case of SUBR (standard functions)
- an S-expression in case of EXPR (user defined functions)
(cf: function type).

F-TYP (abbreviation of Function Type) contains the coded type of the function stored in the F-VAL (cf: funtion type). The following types are available :
- for SUBR : 0SUBR 1SUBR 2SUBR 3SUBR NSUBR FSUBR
- for EXPR : 0EXPR 1EXPR NEXPR LEXPR FEXPR MACRO ESCAPE
The couple F-VAL,F-TYP allows to handle function invocations very quickly (cf: function invocation).

P-TYP (abbreviation of Print Type) contains data needed to edit the external representation of the atom
- as a variable (e.g. replacement of the " or / characters)
- as a function (e.g. specific format to use during the PRETTY-PRINT).

A-LINK (abbreviation of the Atom Link) contains the link to the next atom in the atom area. This link facilatates the hash-coding of the atoms table.

P-LEN (abbreviation of Print Length) contains the size of the external representation of the atom (P-name).

P-NAME (abbreviation of Print Name) contains the characters of the external representation of the atom.

These attributes are stored following the scheme :

```
pointer to atom  →   ┌──────────────────┐
        p + 2    →   │      C-VAL       │
                     │      P-LIST      │
                     ├──────────────────┤
        p + 4    →   │      F-VAL       │
        p + 6    →   │ F-TYP  |  P-TYP  │   ←   p + 7
                     ├──────────────────┤
        p + 8    →   │      A-LINK      │
        p + 10   →   │ P-LEN  |  Pn1    │   ←   p + 11
                     ├──────────────────┤
        p + 12   →   │  Pn2   |  Pn3    │
                     │       ...        │
                     │ PnN-1  |  PnN    │
                     └──────────────────┘
```

All pointers to literal atoms must be a multiple of 4. Then the general form of this kind of pointer is :
01xxxxxx xxxxxx00
This organisation implies a storage requirement of
- 12 bytes for the mono-character atoms

- 16 bytes if the P-LEN has until 5 characters
- 20 bytes if the P-LEN has until 9 characters ...

2.5 Numbers

The management of the numbers is cumbersome in LISP-like interpreters, which are designed to handle pointers, because it is impossible to distinguish a pointer from a number.
Several solutions have been proposed :

- to store each number inside a VLISP Cons-Cell [VLISP 16] containing in the CAR part a special indicator and in the CDR part the value of the number. This representation doubles the storage requirement for the numbers but allows the use of lists management procedures (i.e. the CONS and the Garbage Collection of the lists).
- to store the numbers in a special area managed dynamically [VLISP 10]. This representation allows a quick type test, normal requirement of memory but requires additionnal management of a special area.
- to store the value of the number inside the pointer itself [VLISP 8], which limits the range of possible values because we have to distinguish this kind of values from pointers.

2.5.1 Integer Numbers

An integer number is represented by a pointer to an area which doesn't contain any VLISP objects (i.e. the interpreter area itself, the stack area ...). This pointer contains a value of the form

00sxxxxx xxxxxxxx

in which s is the sign of the number. This representation permits a 14 bit arithmetic and allows to count from

[-2↑13 , 2↑13[i.e. [-8192 , +8192[.

This kind of number doesn't need extra storage and can be tested quickly. Each integer has a unique representation, thus pointers test functions (as EQ, MEMQ ...) can be used for numeric arguments, moreover all interpreter addresses are considered as numbers.
On the other hand, we have to change the representation (by translating 14 bit numbers to 16 bit numbers) before any computation with these numbers, and to use another process to convert all computed values to their VLISP representation. This process (known as number boxing) is quite easy, it must ckeck the 14 bit overflows and clear the two MSB of the value.

2.5.2 Floating point numbers

A floating point number is represented by a pointer to a block of 4 bytes stored in a special area (the floating point numbers area). A value of this kind is coded :

s1 s2 eeeeee mmmmmmmm mmmmmmmm mmmmmmmm

in which s1 is the sign of the mantissa (coded on 24 bits), s2 is the sign of the exponent (on 6 bits). It is possible to compute in the range [2.7x-10↑20 , 9.2x10↑18].
These numbers are available only on the Zilog 80 system and arithmetic functions are handled by the fast arithmetic processing unit Am9511 which performs all floating arithmetic computations.

2.6 Lists

Lists are represented in a standard way : an element of a list (a Cons-Cell) is a pair of pointers, a CAR pointer and a CDR pointer. One Cons-Cell occupies 4 bytes. A pointer to a Cons-Cell is the address of the first byte of the CAR part. All pointers to a Cons-Cell must be a multiple of 4. The general form of this kind of pointer is :

1xxxxxxx xxxxxx00

The access of the CAR and the CDR can be done by the procedures :

```
            ; (CDR HL) → HL   and   (CAR HL) → HL

CDR:        INR     L       ; Skip the CAR part
            INR     L       ;  (two bytes).
CAR:        MOV     A,M     ; A ← MSB of the pointer.
            INR     L       ; Point to LSB.
            MOV     L,M     ; L ← LSB of the pointer.
            MOV     H,A     ; H ← MSB of the pointer.
            RET             ;
```

Obviously, the interpreter doesn't use these procedures but instead uses macro-generation.

To create a new Cons-Cell, a specialized procedure exists which uses the free-list pointer FREEL. Exhaustion of this list can be pointed out in 2 ways :
- by testing explicitly the end of the list
- by trapping (using another bus address demon)

The procedure described uses the first method. When entering in this procedure, HL contains the CAR part of the Cons-Cell to be create and DE contains the CDR part. Exiting the CONS, HL will contain the pointer to the newly allocated Cons-Cell.

```
CONS:       XCHG            ; HL <-> DE.
XCONS:      MOV     B,H     ; Save into the register pair BC,
            MOV     C,L     ;   the CAR part i.e. HL.
            LHLD    FREEL   ; Pick up the free-list pointer.
; if the Garbage Collection is not automaticaly  invoked
; test if end of the free-list
            MOV     A,H     ; FREEL = 0 ? (means
            ORA     L       ;   free-list exhausted)
            CZ      GCOL    ; Yes : call the Garbage-Collection.

            PUSH    HL      ; Save the result of the CONS.
            MOV     M,D     ; Store the CAR part (DE) of
            INR     L       ;   the Cons-Cell
            MOV     M,E     ;   allocated (in HL).
            INR     L
            MOV     D,M     ; Store the CDR part (BC) of
            MOV     M,B     ;   the Cons-Cell
            INR     L       ;   allocated (in HL)
            MOV     E,M     ;   and recover the new
            MOV     M,C     ;   free-list pointer (in DE).
            XCHG
            SHLD    FREEL   ; Save the new free-list pointer.
            POP     HL      ; Recover the address of the
            RET             ;   allocated Cons-Cell.
```

2.7 The Garbage Collection

The Garbage-Collection (G.C.) is a machinery designed to get back the lost Cons-Cells. VLISP 8 uses the stack algorithm which needs two passes : the first one marks all used Cons-Cells and the second sweeps the entire list area, recovering all the non-marked Cons-Cells and removing the marks. The too small address space disables the use of the incremental G.C. [Baker 1977b 1977d].

One problem raised by the G.C. is to find means of marking used Cons-Cells. LISP objects have the following form :

```
00xxxxxx xxxxxxxx for integer numbers
01xxxxxx xxxxxx00 for floating point numbers and literal atoms
1xxxxxxx xxxxxx00 for Cons-Cells.
```

There is no free bit in all types. A bit table management being too cumbersome (the 8080 doesn't have intructions on bit), the G.C. mark uses the less significant bit (LSB) of the CDR part of each Cons-Cell. This bit is always 0 unless in the case of integer numbers. For this new implementation problem, two solutions appear :
- change the internal representation of integer numbers in such way that the LSB is always 0. This constrains to shift the value of the numbers and so to loose half of the available values but above all to prevent to consider addresses in the interpreter as numbers.
- forbide the creation of Cons-Cells whose CDR part is an integer number.

At present, the second solution is implemented in all the systems, and an error occurs during the CONS of such a Cell. Thus the mark bit can be the LSB of the CDR part.

3.0 The INTERPRETER

In order to speed up evaluations of forms a new user-function invocation has been designed and the function interpreter doesn't perform any CONS (by the use of EVLIS function) unless explicitly asked for LEXPR type functions.

3.1 Access and binding of variables

The use of A-list to handle dynamic variable bindings has been given up in all VLISP systems.
The variable binding uses the shallow-binding method [Baker 1977a], whose main charasteristic is to associate each variable to a unique location containing, at any time, its actual value.
The access becomes as fast as with a static binding (à la FORTRAN). On the other hand the environment must be saved when entering and restored when exiting from a function.
A great lack of such a binding : FUNARG problems [Moses 1970] are not correctly handled.

To realize this type of binding, we have to :
- allocate for every atom a unique location which always contains its value : the C-Value
- save on the stack, when entering the function, the current value of each formal parameter.
- bind actual values of formal parameters to their C-Values.
- restore from the stack, when exiting the function, the old values of all actual parameters.

For each function invocation, a control-block is build on the stack. The form of this block is :

```
[     variable 1          ]   ← P$BIND
[  old value of var 1     ]
[     variable 2          ]
[  old value of var 2     ]
         ...
[     variable N          ]
[  old value of var N     ]
[*** end block mark ****]
[    old P$BIND           ]
```

These blocks themselves are linked through the P$BIND link. This link is necessary to interpret LESCAPE, ESCAPE and SELF functions. Therefore the environment share [Bobrow 1973] is not allowed.

To illustrate the variables access, here is the begining of the main function of the interpreter, the EVAL function :

```
             ; (EVAL HL)  →  HL

EVAL:   MOV     A,H       ; A ← MSB of pointer in HL
        CPI     HVAR      ; Direct return in case of
        RC                ; integer or floating point numbers,
                          ; and literal constants (NIL T ...)
        CPI     HLIST     ; Is it a pointer to a Cons-Cell ?
        JNC     EVALF     ; Yes : go handle list forms.
        MOV     D,M       ; DE ← C-Val of HL.
        INR     L
        MOV     E,M
        XCHG              ; HL ← C-val
        MOV     A,H       ; A ← MSB of this value.
        CPI     HUNDF     ; Is it defined ?
        RNZ               ; Yes.
        DCR     E         ; No : DE contains the name
                          ; of the undefined variable.
```

Evaluation of numbers or literal constants spends 23 clock cycles, and evaluation of variables, 80 clock cycles.

3.2 Function types

There are two families of functions :
- SUBR functions written in the 8080 machine language and executed directly by the micro-processor.
- EXPR functions written in VLISP and interpreted by the evaluator.

In each family a distinction is made between functions with 0, 1, 2, 3 or N evaluated arguments (type function name is the family name prefixed by 0, 1, 2, 3 or N, ex: NSUBR 1EXPR ...) and the functions with N non-evaluated arguments (type function name is the name of the family prefixed by F : FSUBR and FEXPR). There are three more special interpreted functions
LEXPR: with any number of arguments, all evaluated and bound to one variable.
MACRO: to generate code which will be evaluated a new time by the interpreter.
ESCAPE: to handle dynamically defined escape functions.

3.3 Definition and invocation of user functions

In the same manner as for the variable values, VLISP 8 does not use the P-list of the atom to store function definitions but uses instead a fixed unique location for each atom (the F-Val) containing at any time, the function associated to this atom. This organisation allows a faster response than a search of an undetermined length list. Associated with this F-Val each atom owns another fixed location which contains the coded type of this function (in order to facilitate the indexed indirect branching on the type of the function, performed by the functions evaluator).
In case of redefinition using definition functions (like DE, DF, DM, ESCAPE and WITH), the couple (F-Val F-Typ) of the previous definition is saved in the P-list under the FUNCTION indicator. This P-list is used like a definition stack permitting a true dynamic redefinition of every function (even SUBR).
Some new functions have been added to manipulate these new attributes, the GETFN function which gets the FVAL-FTYP attribute couple and SETFN which sets this couple. The P-list manipulation is handled by standard functions GETPROP, ADDPROP, PUTPROP and REMPROP.

Function invocations are very powerful and versatile.
If a function call has too few arguments, they have the defaut value NIL (these arguments can be considered as local variables), on the other hand, extraneous arguments are ignored without evaluating them (VLISP 8 don't allow side effect during arguments evaluation).
Morever if a function is undefined (the F-Val is NIL) an indirection by the C-Val of the atom (if that C-Val is not a constant) is performed.

VLISP 8 includes recent developments of others VLISP systems.
Particularly, in order to replace the PROG feature which has been removed (for speed reasons) the following functions have been implemented :
LESCAPE which allows to RETURN from any Lambda-expression
ESCAPE which allows to dynamically define escape functions. These functions are used as super-RETURN. ESCAPE is the functionnal equivalent of the CATCH an THROW of MACLISP.
SELF invokes the last called function and permits to solve the LABEL functions problem.

Some cases of recursion such as the tail-recursion are interpreted iteratively [Greussay 1976b]. For exemple this toplevel loop leaves the stack unchanged :

```
(DE TOPLEVEL ()
   (PRINT (EVAL (READ)))
   (TOPLEVEL))
```

CONCLUSIONS :

The actual limits of this system are :

1) the speed of the interpreter : with clocks working at 2.5 MHz (thus with a base cycle of 400nS) and with instructions manipulating mainly bytes, performances exceed widely those of old 16 bits mini-computers (as T1600, MITRA 15 ...) but are obviously worse than those of big computers such as the PDP10.

2) - the address space : memory address is handled with 16 bits and the memory is organized in 8 bit words, so the total addressable space is of 64k byte. With the maximum configuration, the number of Cons-Cells does not exeed 8k which is not suffisant to store important VLISP programms such as PHENARETE [Wertz 1978], CAN [Goossens 1977] or even the compiler.

The arrival of the 16 bit micro-processors (for example the Intel 8086 or the Zilog 8000) will allow to increase the performances of systems such as ours :

1) - for the speed : clocks working up to 4MHz (with a base cycle of 250nS), and instruction sets allowing to manipulate large data of 16, 32 or even 64 bits, will reduce

2) - for the address space, with address bus forseeds up to 24 bits, allowing to address up to 8M byte (that implies 2M Cons-Cells).

In spite of these limitations, VLISP 8 is widely used by different research groups of our University, especially the "Groupe Art et Informatique" using largely the possibility to program specialized terminals, and the "Groupe LOGO-LISP", which has written a MICRO-LOGO-LISP [WERTZ 1979] and appreciates highly the portability of such a system making it a very powerful pedagogical tool.

4.0 References

ALLEN J. : The Anatomy of LISP, Mc Graw Hill, 1978.

AUDOIRE L. : COLORIX : un périphérique de visualisation couleur, Mémoire de maîtrise, U.E.R. Informatique, Univresité de Paris VIII, Juin 1976.

BAKER Jr, H.G. : Shallow binding in LISP 1.5, M.I.T. Artificial Intelligence Laboratory, Working Paper 138, January 1977.

BAKER Jr, H.G. : A Note On the Optimal Allocation of Space in MACLISP, M.I.T. Artificial Intelligence Laboratory, Working paper 142, March 1977.

BAKER Jr,H.G. and HEWITT C. : The incremental Garbage Collection of Processes, M.I.T. Artificial Intelligence Laboratory, Working Paper 149, June 1977.

BOBROW D.G., WEGBREIT B. : A Model of Stack Implementation of Multiple Environments, C.A.C.M. vol 16 nb 10, October 1973.

CHAILLOUX J. : VLISP 10 . 3 , Manuel de Références, Université de Paris VIII - Vincennes, Août 1978.

CHAILLOUX J. : a VLISP interpreter on the VCMC1 machine, LISP Bulletin #2, July 1978.

GOOSSENS D. : CAN, Département d'Informatique, Université de Paris VIII - Vincennes, 1977.

GREUSSAY P. : Manuel LISP 510 : description et utilisation, Institut d'Intelligence Artificielle, Université de Paris VIII - Vincennes, NTP 2, Octobre 1972.

GREUSSAY P. : Iterative interpretations of tail-recursive LISP procedures, Département d'Informatique, TR 20-76, Université de Paris VIII - Vincennes, Septembre 1976.

GREUSSAY P. : Contribution à la définition interprétative et à l'implémentation des LAMBDA-langages, Thèse, Université de Paris VII, Novembre 1977.

GREUSSAY P. : VLISP 16, Manuel de Référence, Ecole Polytechnique, Decembre 1978.

McCARTHY J. et al. : LISP 1.5 Programmer's manual, the M.I.T. Press, Cambridge, Mass., 1974.

MOSES J. : the function of FUNCTION in LISP, Memo 199, M.I.T. Artificial Intelligence Laboratory, June 1970.

STEELE G.L. : BIBOP LISP, M.I.T. Artificial Intelligence Laboratory, Mars 1974.

WERTZ H. : LISP CAB500, Rapport de recherche groupe II équipe 9, Université de Paris VIII - Vincennes, 1975.

WERTZ H. : Un système de compréhension, d'amélioration et de correction de programmes incorrects, Thèse de 3ème cycle, Université Paris VI, Juillet 1978.

WERTZ H. : Computer Aided Education for Mentally Retarded Children, Proc. Int. Symposium on Computers and Education, March 1979, Dusseldorf, RFA.